Fighting latency

How to optimize your system using perf

Mischa Jonker

October 24th, 2013

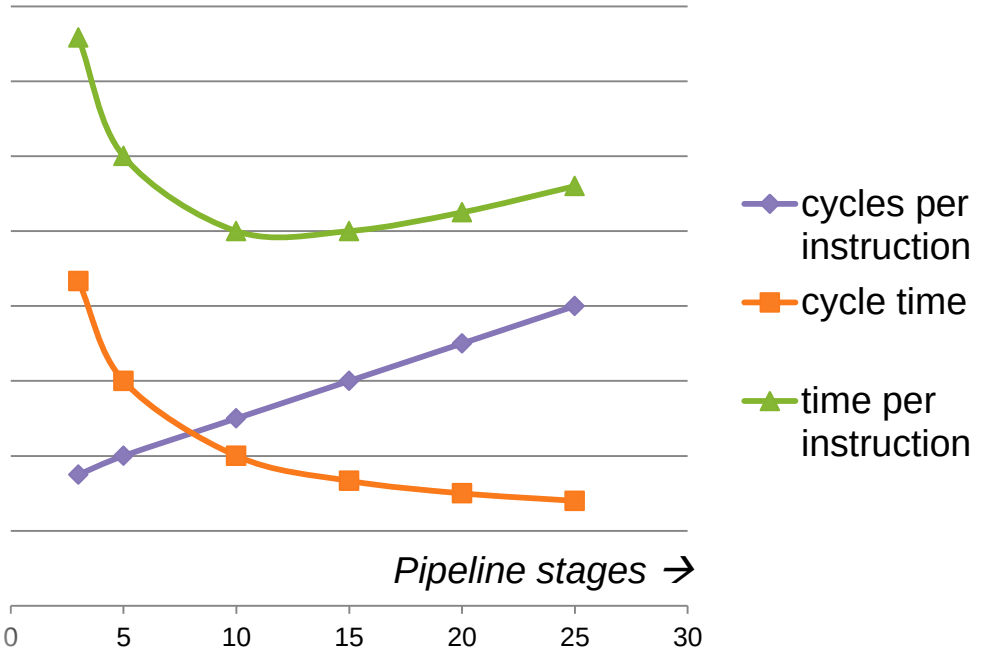
Contents

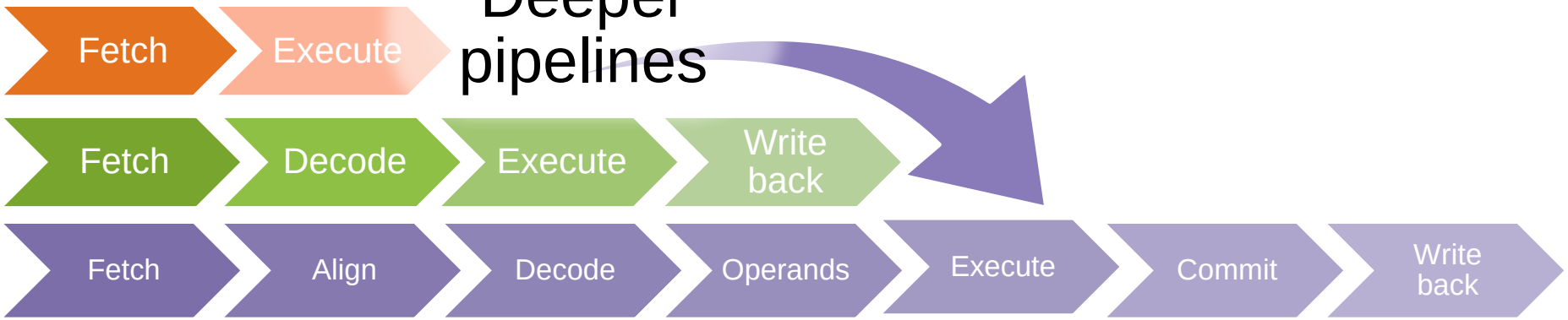
- Introduction
  - Processor trends; what kind of latency are we fighting?
- What is perf?
- Using perf to identify bottlenecks
- Prefetching
- Using GCC options to tune prefetching

Processor trends

Old problems, but now in embedded CPU's

- To get more performance, processors get deeper pipelines
  - Split the work load in multiple stages, so time per cycle gets shorter



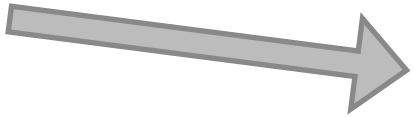


Causes of a high CPI

Example: simplified memcpy loop in assembly

- The branch at the end of the loop is predicted taken, so the CPU can keep on filling pipeline stages

```
1: ld r1, [r2]
sub.f r0, r0, 1
st r1, [r3]
add r2, r2, 4
add r3, r3, 4
bnz 1b
<do something else>
```



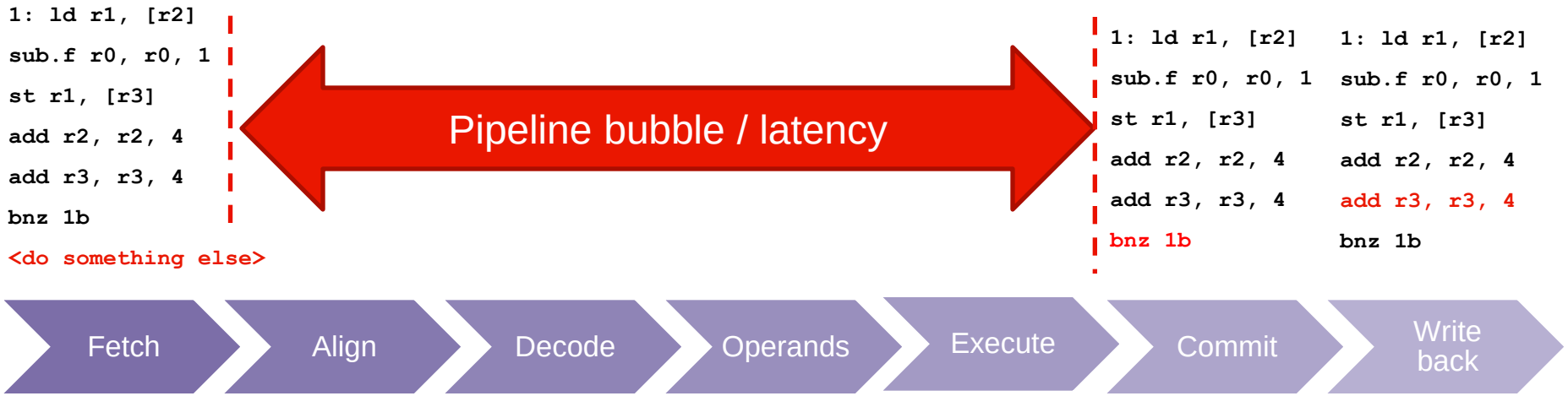
1: ld r1, [r2]	1: ld r1, [r2]	1: ld r1, [r2]	1: ld r1, [r2]	1: ld r1, [r2]	1: ld r1, [r2]	1: ld r1, [r2]
sub.f r0, r0, 1	sub.f r0, r0, 1	sub.f r0, r0, 1	sub.f r0, r0, 1	sub.f r0, r0, 1	sub.f r0, r0, 1	sub.f r0, r0, 1
st r1, [r3]	st r1, [r3]	st r1, [r3]	st r1, [r3]	st r1, [r3]	st r1, [r3]	st r1, [r3]
add r2, r2, 4	add r2, r2, 4	add r2, r2, 4	add r2, r2, 4	add r2, r2, 4	add r2, r2, 4	add r2, r2, 4
add r3, r3, 4	add r3, r3, 4	add r3, r3, 4	add r3, r3, 4	add r3, r3, 4	add r3, r3, 4	add r3, r3, 4
bnz 1b	bnz 1b	bnz 1b	bnz 1b	bnz 1b	bnz 1b	bnz 1b

Causes of a high CPI (2)

Example: simplified memcpy loop in assembly

– If the branch is not taken / mispredicted, the pipeline needs to be **flushed** and a different instruction needs to be fetched!

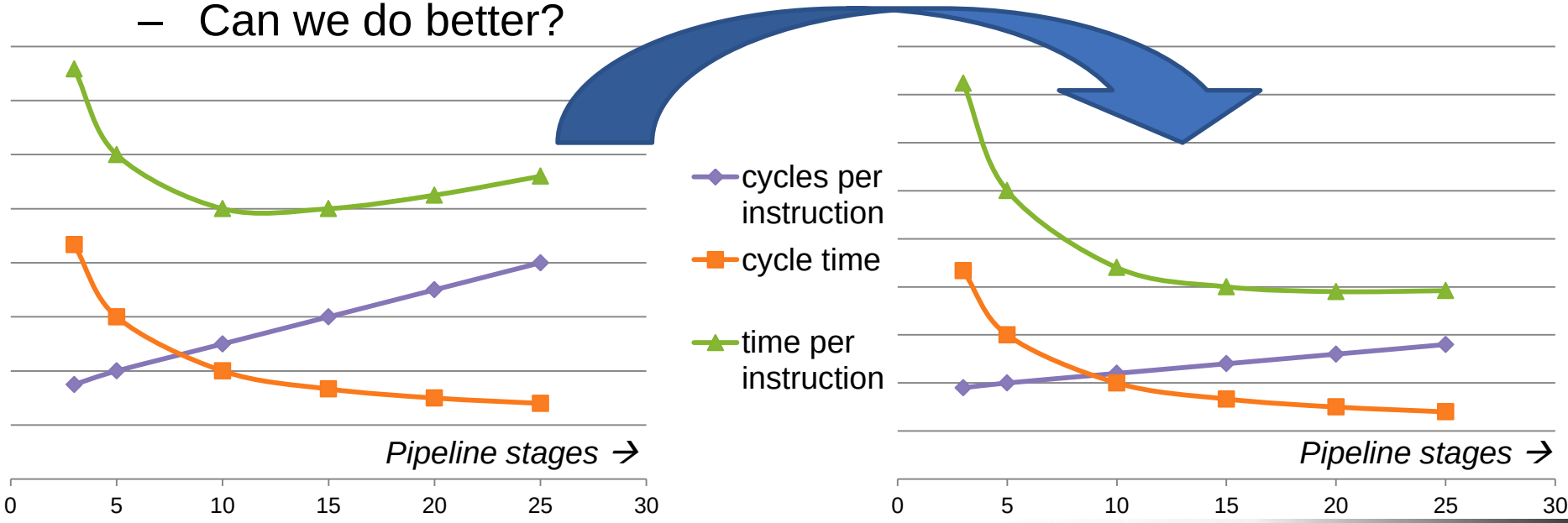
```
1: ld r1, [r2]
sub.f r0, r0, 1
st r1, [r3]
add r2, r2, 4
add r3, r3, 4
bnz 1b
<do something else>
```



Processor trends

How to keep CPI low?

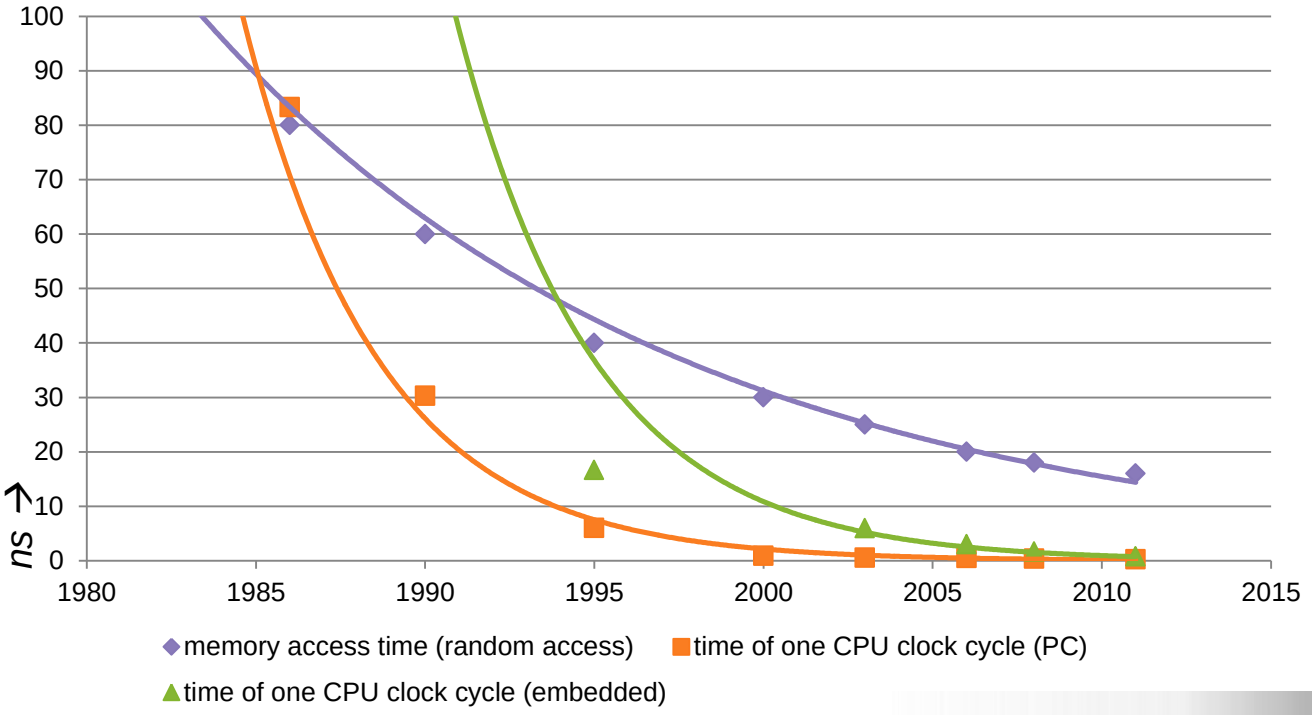
- Various ways to keep CPI low:
  - Do multiple instructions at once (super-scalar)
  - To decrease the penalty of branch mispredicts, we can speculatively start with execution of both paths;
- However, this costs power and area (# of transistors)
  - Can we do better?



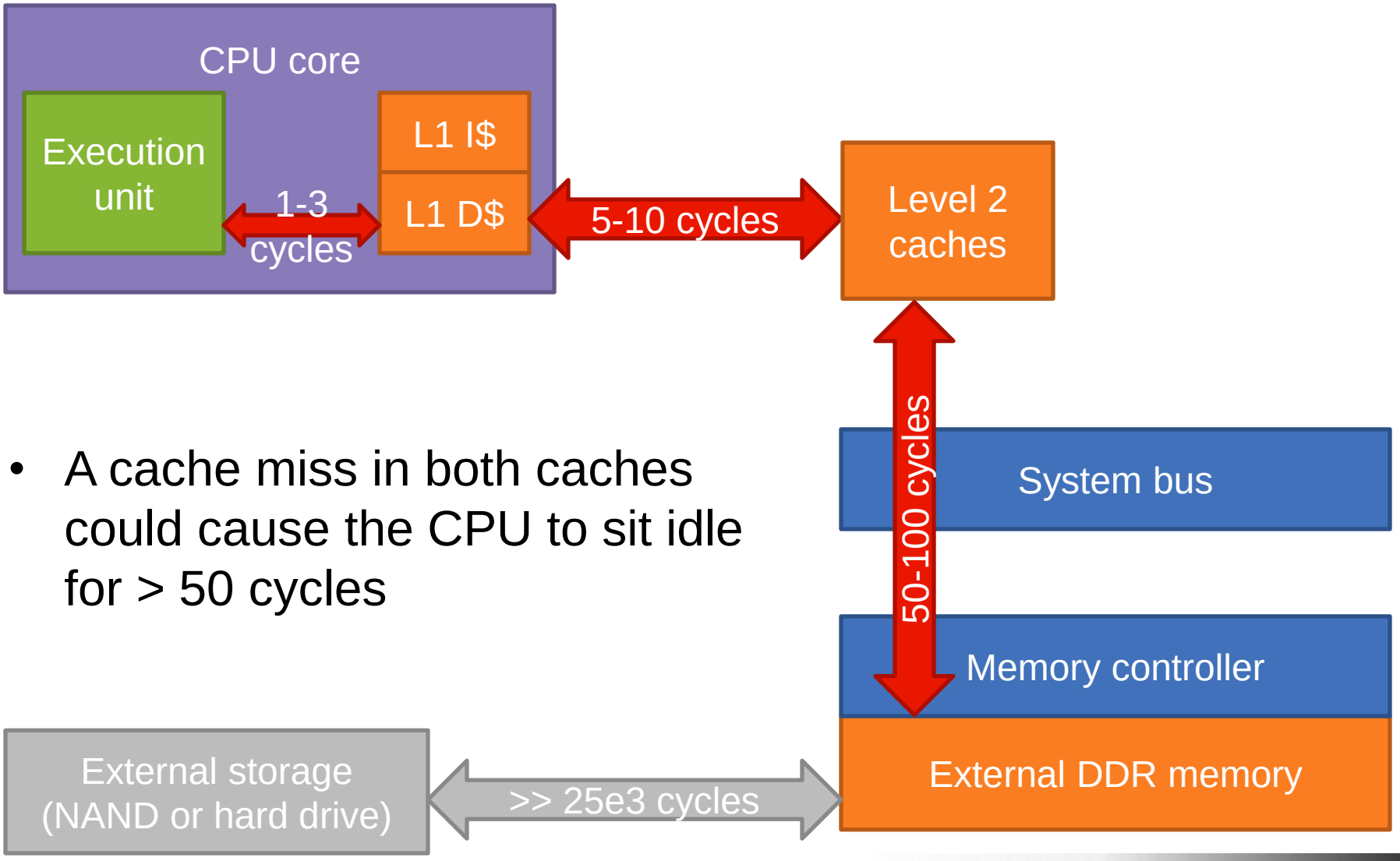
Memory latency

Old problems, but now in embedded CPU's (2)

- Memory latency is decreasing, but CPU speeds are increasing at a faster rate
  - Now memory is also bottleneck for embedded CPU's
  - Latency increases further with multiple cores



Memory latency



- A cache miss in both caches could cause the CPU to sit idle for > 50 cycles

What is perf?

- Originally Performance Counters for Linux (PCL)
  - Counts HW events (cache misses, pipeline stalls, etc.)
  - Uses kernel infrastructure, **no instrumentation required, low overhead**
  - Renamed to perf events in 2009 when it became more generic
- Used to optimize the software for the ATLAS detector that found the Higgs particle

What is perf?

- Two modes:
  - Statistics: just count events
  - Profiling: for every n^{th} event, record PC

Needs root!
include/trace/events/*.h for examples

HW events	SW events	Trace points (needs root!)
• Stall cycles • D$ misses • TLB reloads • Branch mispredicts	• Page faults • Context switches • Clock (interval timer)	• Specific system calls • Various file system hooks • Etc.

How to use perf?

- Prerequisites:
  - Perf tools in your rootfs
    - For instance, using Buildroot, enable **BR2_PACKAGE_PERF**
  - Kernel with Perf enabled
    - Enable **CONFIG_PERF_EVENTS**
    - For trace points, **CONFIG_TRACEPOINTS** needs to be enabled. This is selected through various kernel config option combinations:
      - **CONFIG_FTRACE** and **CONFIG_FUNCTION_TRACER;**
      - **CONFIG_FTRACE** and **CONFIG_ENABLE_DEFAULT_TRACERS;**
      - **CONFIG_KPROBE_EVENT**

How to use perf?

- It has a git-like command interface
- To just get statistics, without profile, you can use:

$ perf stat <*command*>

```
# perf stat echo hello world
hello world

 Performance counter stats for 'echo hello world':

          4.000000 task-clock                #    0.784 CPUs utilized
                 3 context-switches          #    0.750 K/sec
                 0 cpu-migrations            #    0.000 K/sec
                 0 page-faults               #    0.000 K/sec
           2455962 cycles                    #    0.614 GHz
            121768 stalled-cycles-frontend   #    4.96% frontend cycles idle
            980344 stalled-cycles-backend    #   39.92% backend  cycles idle
            301142 instructions              #    0.12  insns per cycle
                                             #    3.26  stalled cycles per insn
             44250 branches                  #   11.063 M/sec
              7702 branch-misses             #   17.41% of all branches

       0.005103600 seconds time elapsed

# 
```

How to use perf?

- For profiling, you need to actually record samples to a file; this is done using:

$ perf record <*command*>

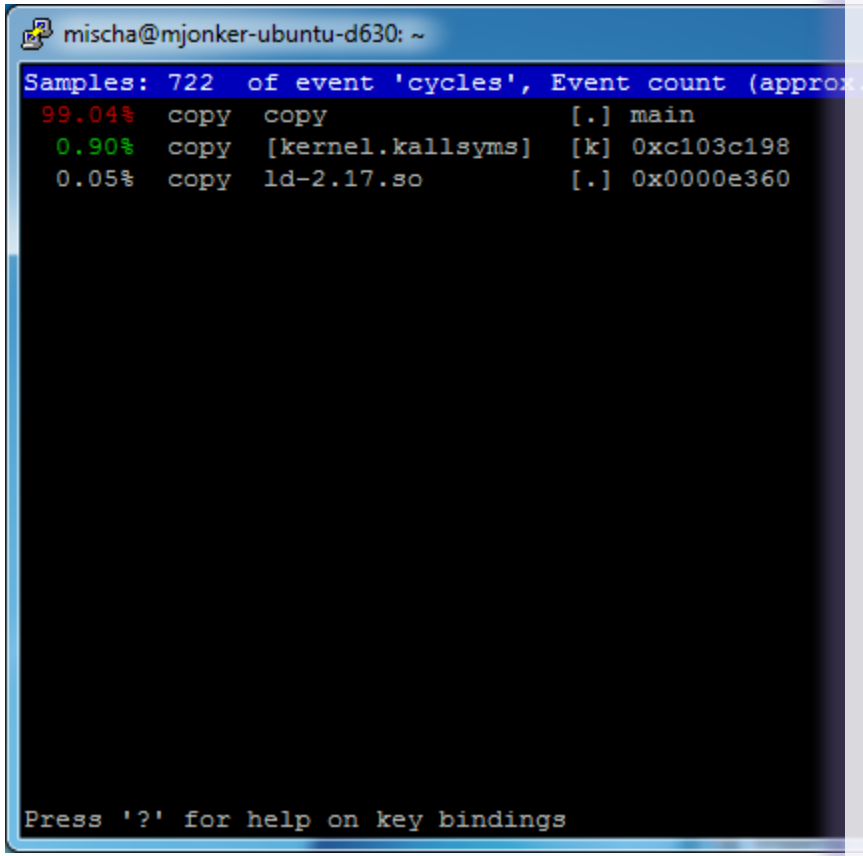
```
mischa@mjonker-ubuntu-d630:~$ perf record ./copy
WARNING: Kernel address maps (/proc/{kallsyms,modules}) are restricted,
check /proc/sys/kernel/kptr_restrict.

Samples in kernel functions may not be resolved if a suitable vmlinux
file is not found in the buildid cache or in the vmlinux path.

Samples in kernel modules won't be resolved at all.

If some relocation was applied (e.g. kexec) symbols may be misresolved
even with a suitable vmlinux or kallsyms file.

[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.030 MB perf.data (~1298 samples) ]
```

How to use perf?

- The result can be obtained with:

```
$ perf report > file.txt
```

```
$ perf report
```

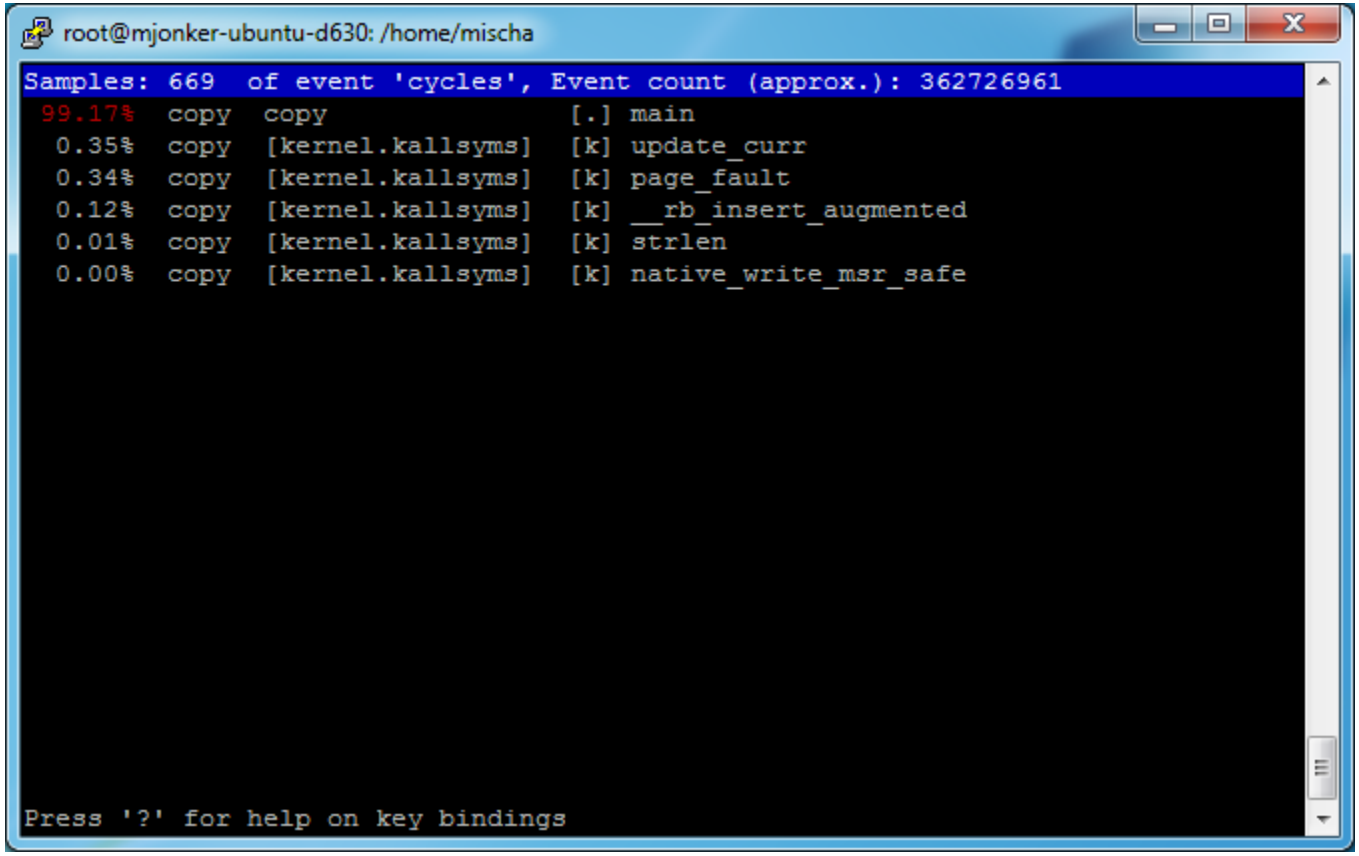



```
mischa@mjonker-ubuntu-d630:~$ cat file.txt
# ========
# captured on: Thu Oct 10 11:45:44 2013
# hostname : mjonker-ubuntu-d630
# os release : 3.8.0-31-generic
# perf version : 3.8.13.8
# arch : i686
# nrcpus online : 2
# nrcpus avail : 2
# cpudesc : Intel(R) Core(TM)2 Duo CPU T7250 @ 2.00GHz
# cpuid : GenuineIntel,6,15,13
# total memory : 2055272 kB
# cmdline : /usr/bin/perf_3.8.0-31 record ./copy
# event : name = cycles, type = 0, config = 0x0, config1 =
0x0, config2 = 0x0, excl_usr = 0, excl_kern = 0, excl_host =
0, excl_guest = 1, precise_ip = 0, id = { 25, 26 }
# HEADER_CPU_TOPOLOGY info available, use -I to display
# pmu mappings: cpu = 4, software = 1, tracepoint = 2,
breakpoint = 5
# ========
#
# Samples: 722  of event 'cycles'
# Event count (approx.): 381229490
#
# Overhead  Command      Shared Object     Symbol
# ........  .......  .................  ........
#
    99.04%     copy  copy               [.] main
     0.90%     copy  [kernel.kallsyms]  [k] 0xc103c198
     0.05%     copy  ld-2.17.so         [.] 0x0000e360
```

How to use perf?

- To enable kernel symbol resolution, you can do the following (as root!!) before starting **`perf record`**

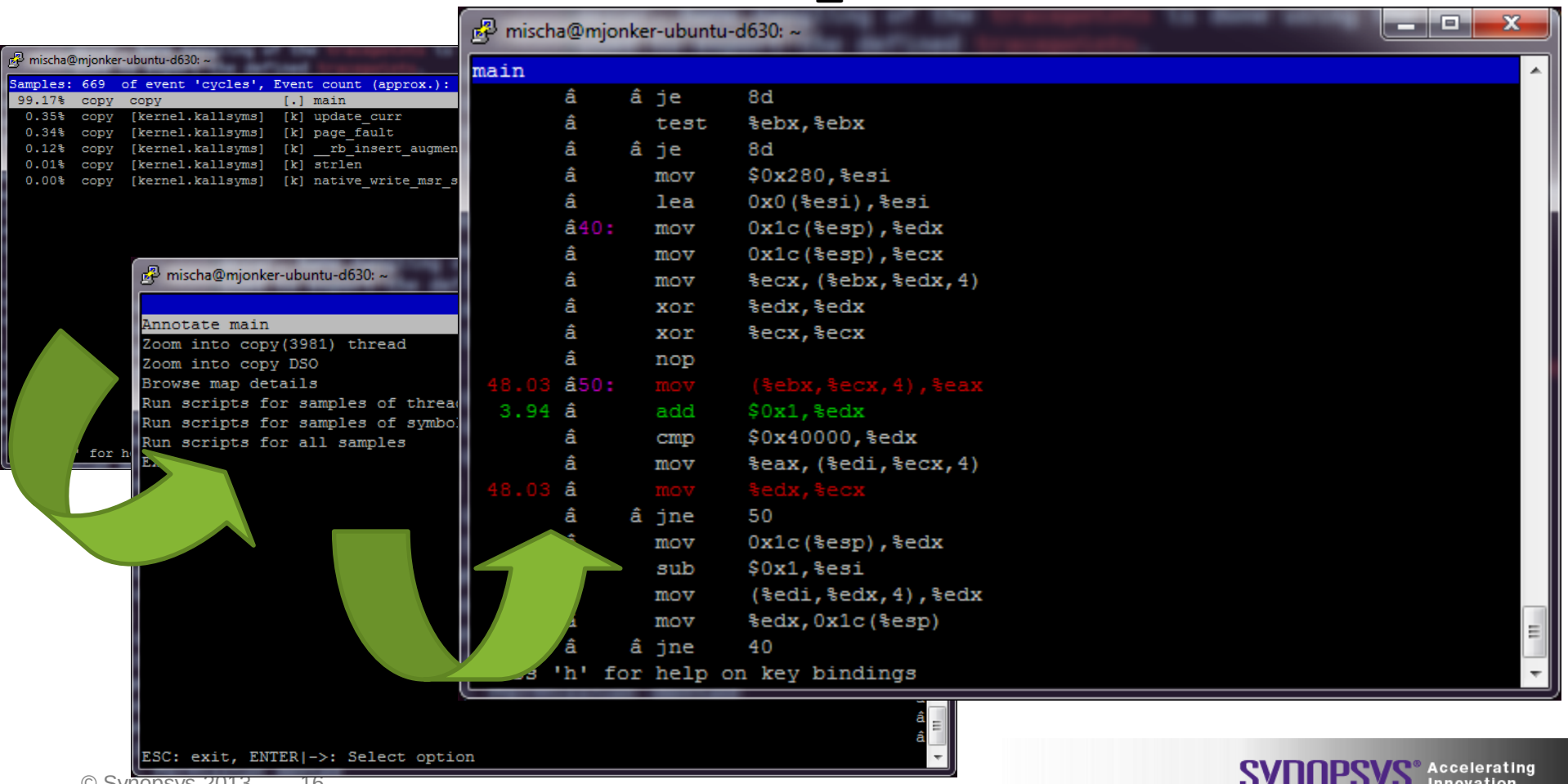
```
# echo 0 > /proc/sys/kernel/kptr_restrict
```



How to use perf?

- To enable kernel symbol resolution, you can do the following (as root!!) before starting **`perf record`**

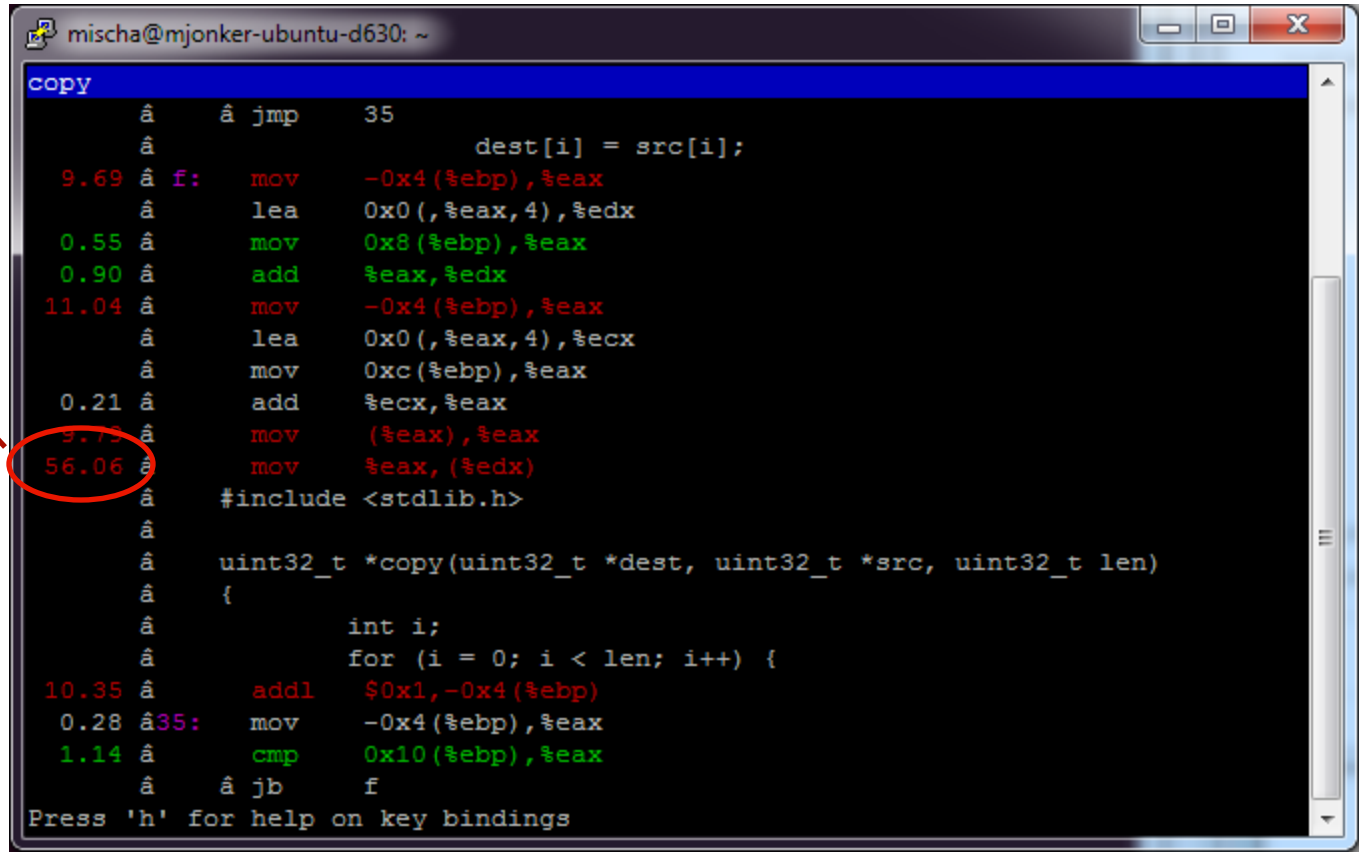
```
# echo 0 > /proc/sys/kernel/kptr_restrict
```



How to use perf?

- To get a better idea of what C code is responsible, compile your program with –O0 –g
  - That's intrusive though

> 50% of cycles spent in one instruction!



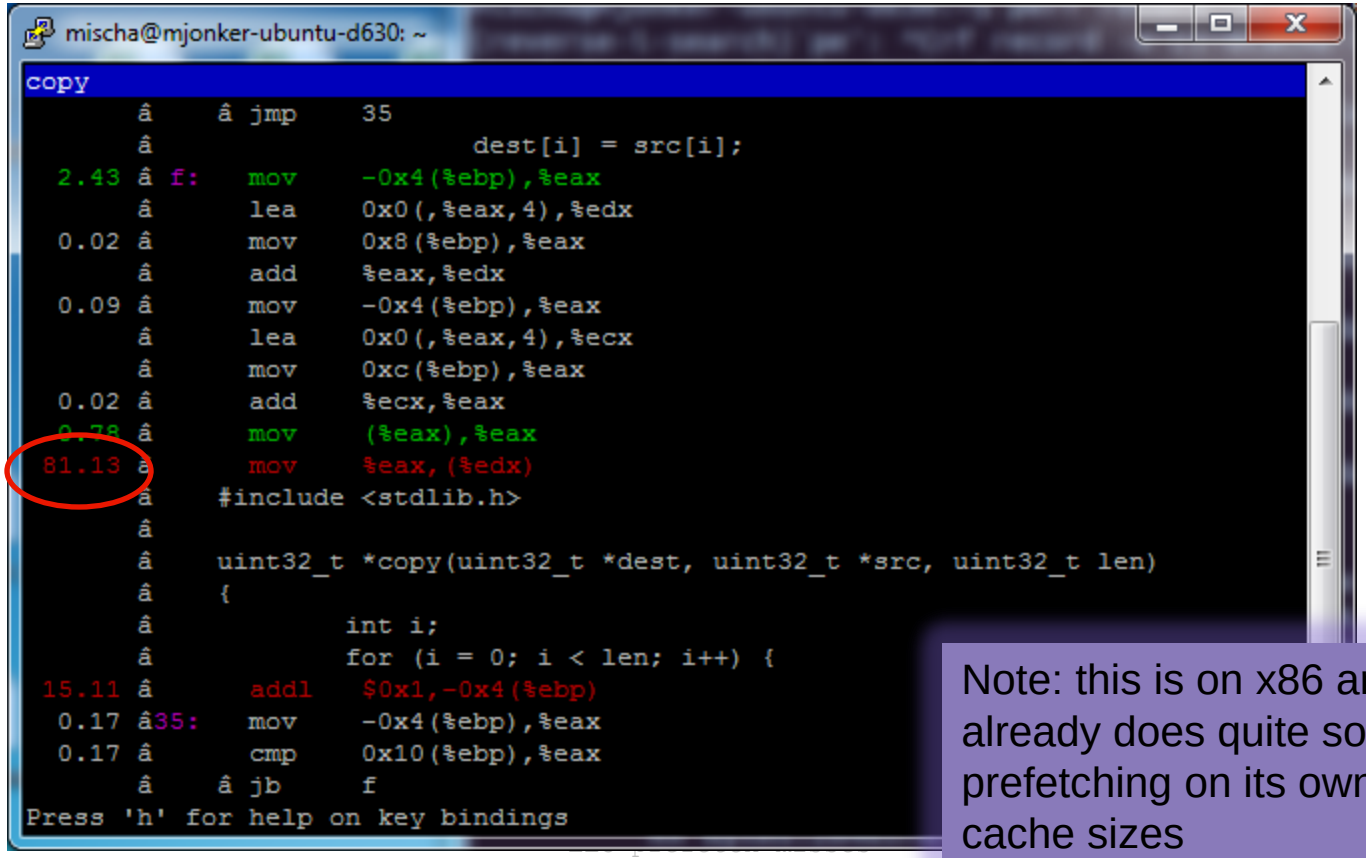
How to use perf?

- By default, perf uses the ‘cycles’ event, with sampling frequency = 4 kHz
- We can use 100’s of different events for sampling;
  - e.g. to trigger a sample for every *n*th D$ load miss, record like this:
    - **`perf record -e L1-dcache-load-misses -c n <command>`**
  - Use **`perf list`** to get a list of events
- Note that cache misses are not time-based events:
  - if a frequency is specified, the frequency is used as a guideline to determine the sampling interval.

```
mischa@mjonker-ubuntu-d630:~$ perf list

List of pre-defined events (to be used in -e):
  cpu-cycles OR cycles                               [Hardware event]
  instructions                                       [Hardware event]
  cache-references                                   [Hardware event]
  cache-misses                                       [Hardware event]
  branch-instructions OR branches                    [Hardware event]
  branch-misses                                      [Hardware event]
  bus-cycles                                         [Hardware event]
  stalled-cycles-frontend OR idle-cycles-frontend    [Hardware event]
  stalled-cycles-backend OR idle-cycles-backend      [Hardware event]
  ref-cycles                                         [Hardware event]

  cpu-clock                                          [Software event]
  task-clock                                         [Software event]
  page-faults OR faults                              [Software event]
  context-switches OR cs                             [Software event]
  cpu-migrations OR migrations                       [Software event]
  minor-faults                                       [Software event]
  major-faults                                       [Software event]
  alignment-faults                                   [Software event]
  emulation-faults                                   [Software event]

  L1-dcache-loads                                    [Hardware cache event]
  L1-dcache-load-misses                              [Hardware cache event]
  L1-dcache-stores                                   [Hardware cache event]
  L1-dcache-store-misses                             [Hardware cache event]
  L1-dcache-prefetches                               [Hardware cache event]
  L1-dcache-prefetch-misses                          [Hardware cache event]
  L1-icache-loads                                    [Hardware cache event]
  L1-icache-load-misses                              [Hardware cache event]
  L1-icache-prefetches                               [Hardware cache event]
  L1-icache-prefetch-misses                          [Hardware cache event]
  LLC-loads                                          [Hardware cache event]
  LLC-load-misses                                    [Hardware cache event]
  LLC-stores                                         [Hardware cache event]
  LLC-store-misses                                   [Hardware cache event]
  LLC-prefetches                                     [Hardware cache event]
  LLC-prefetch-misses                                [Hardware cache event]
  dTLB-loads                                         [Hardware cache event]
  dTLB-load-misses                                   [Hardware cache event]
  dTLB-stores                                        [Hardware cache event]
  dTLB-store-misses                                  [Hardware cache event]
  dTLB-prefetches                                    [Hardware cache event]
```

How to use perf?

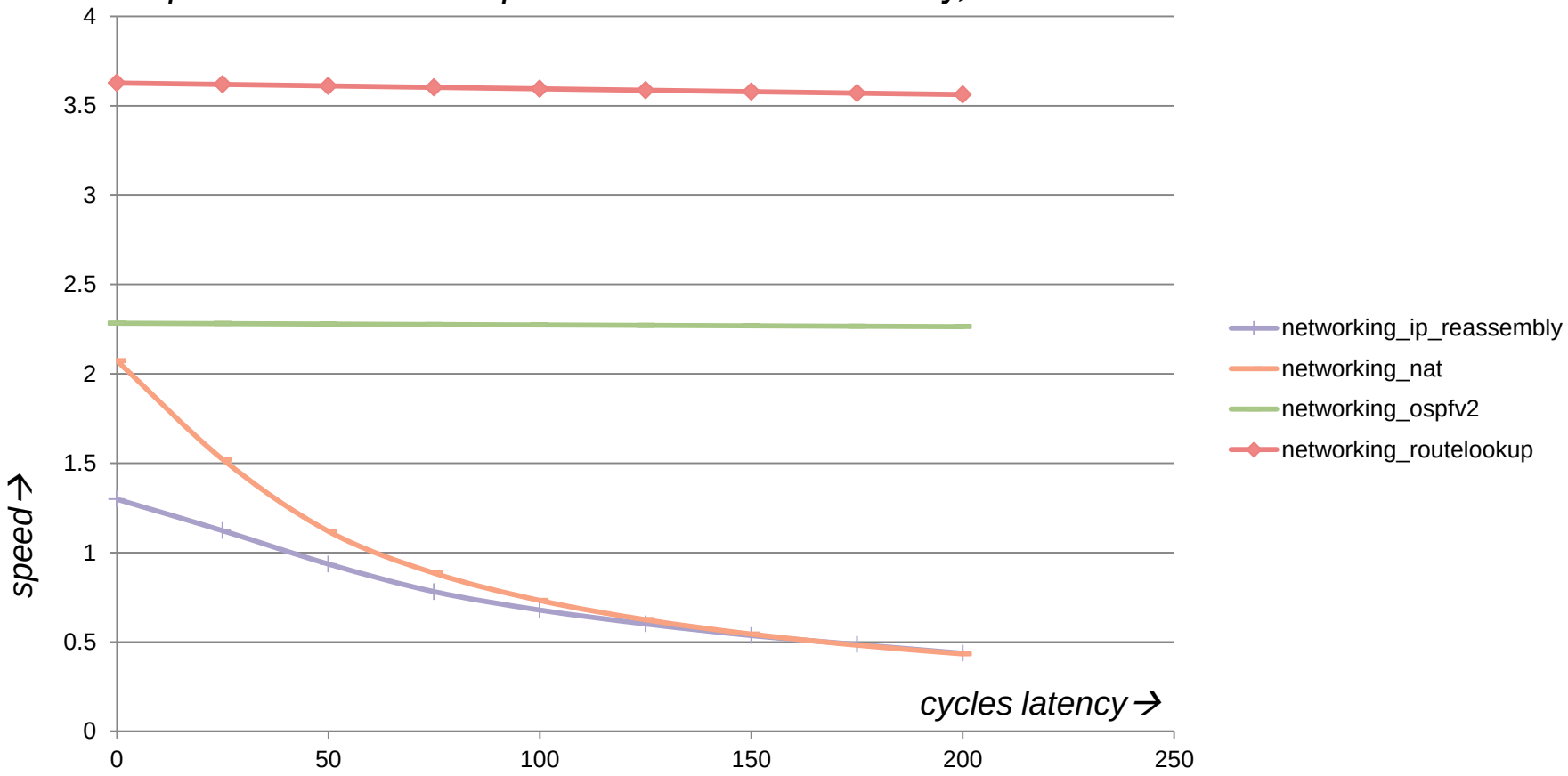
- Looking at L1 D$ load misses:
  - one instruction responsible for > 80% of D$ ld misses!



Note: this is on x86 architecture, which already does quite some speculative prefetching on its own, and has large cache sizes

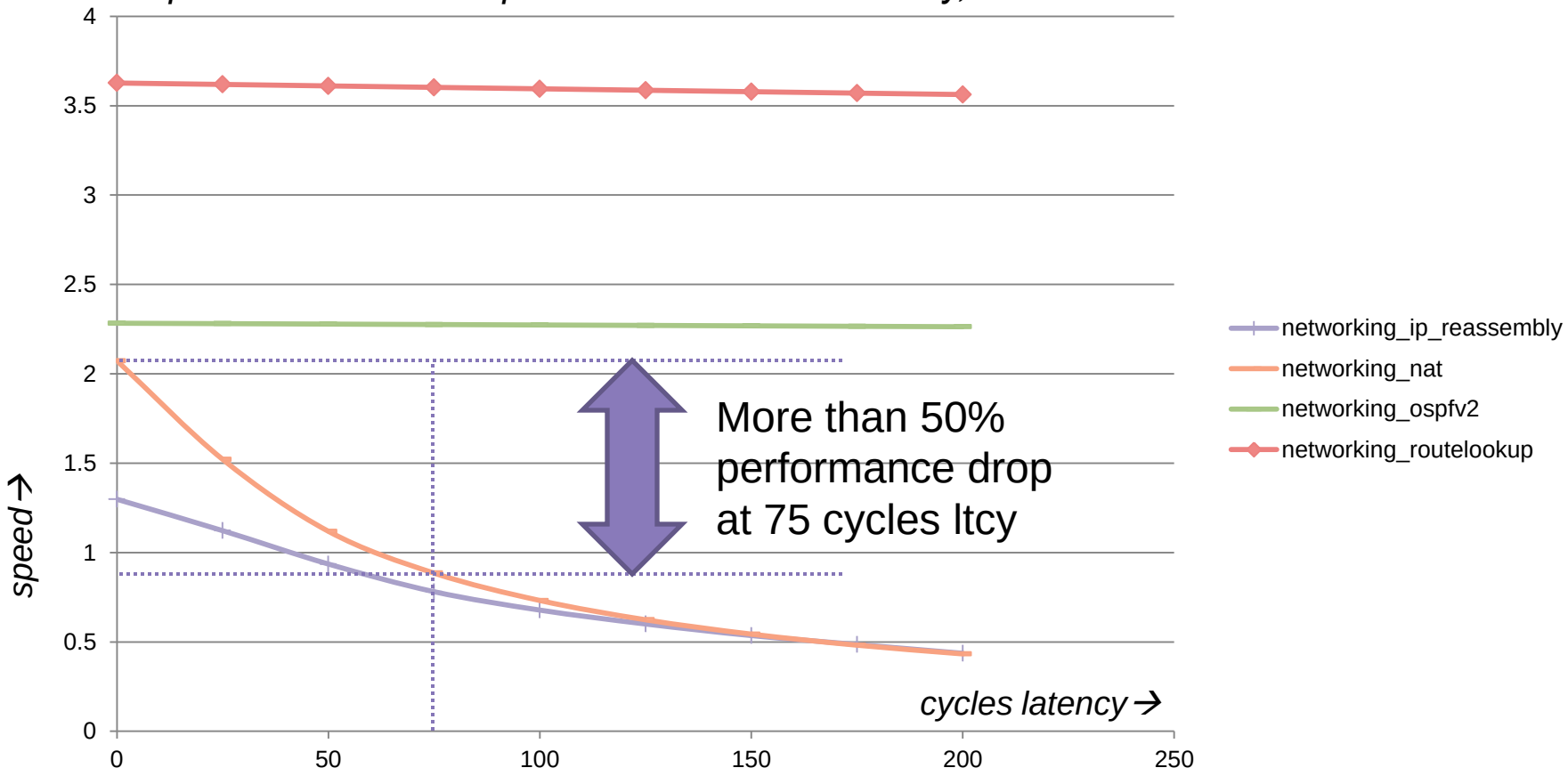
Need for prefetching

- Performance of any benchmark that uses data sets >> D$ size drops (dramatically) when memory latency increases
- *Example: Network benchmarks (Iterations/s/MHz for various memory latencies), ospfv2 and routelookup don't use a lot of memory, the other two do*



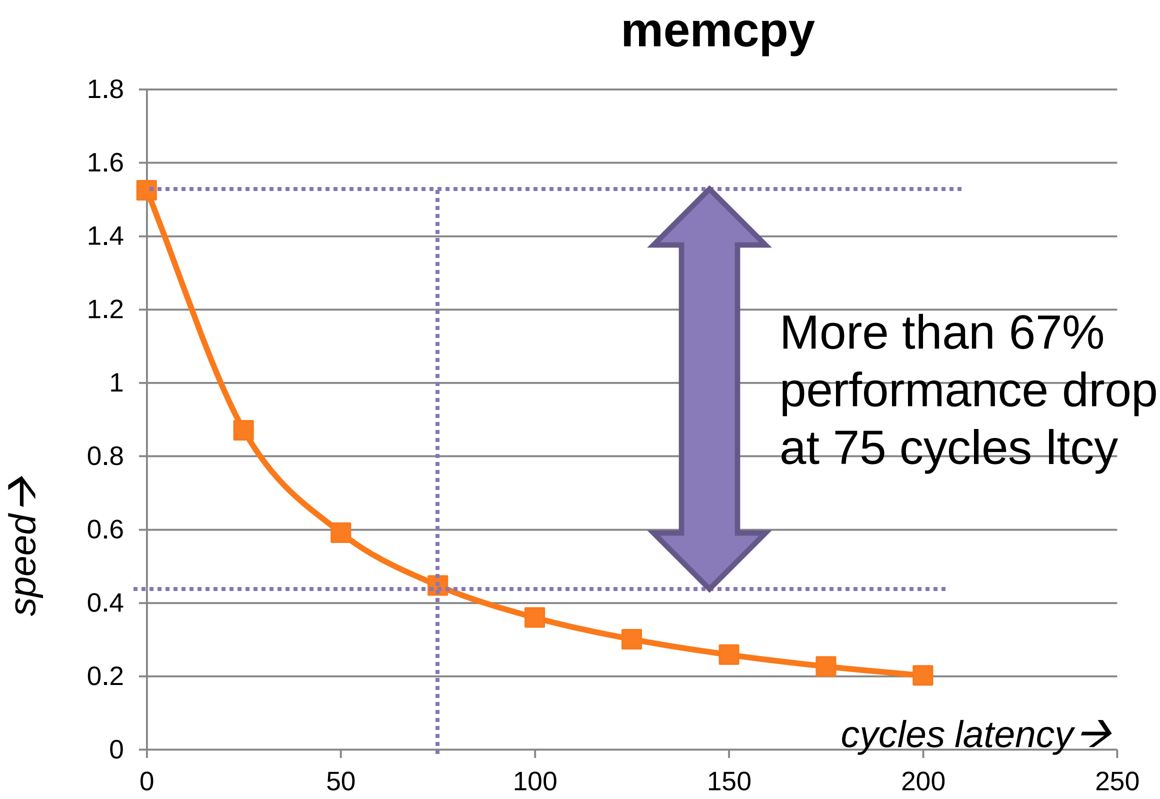
Need for prefetching

- Performance of any benchmark that uses data sets >> D$ size drops (dramatically) when memory latency increases
- *Example: Network benchmarks (Iterations/s/MHz for various memory latencies), ospfv2 and routelookup don't use a lot of memory, the other two do*

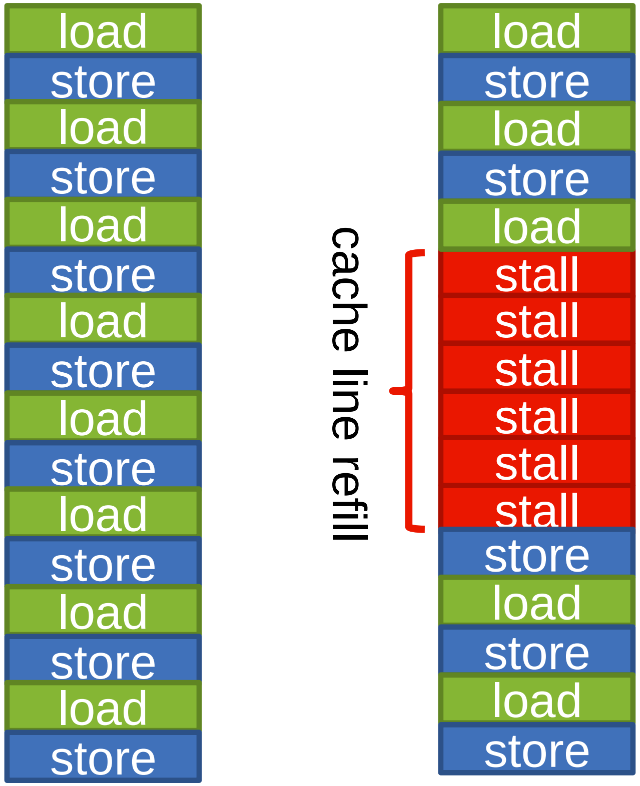


Need for prefetching

- Plain memcpy shows even more performance degradation with increasing memory latency

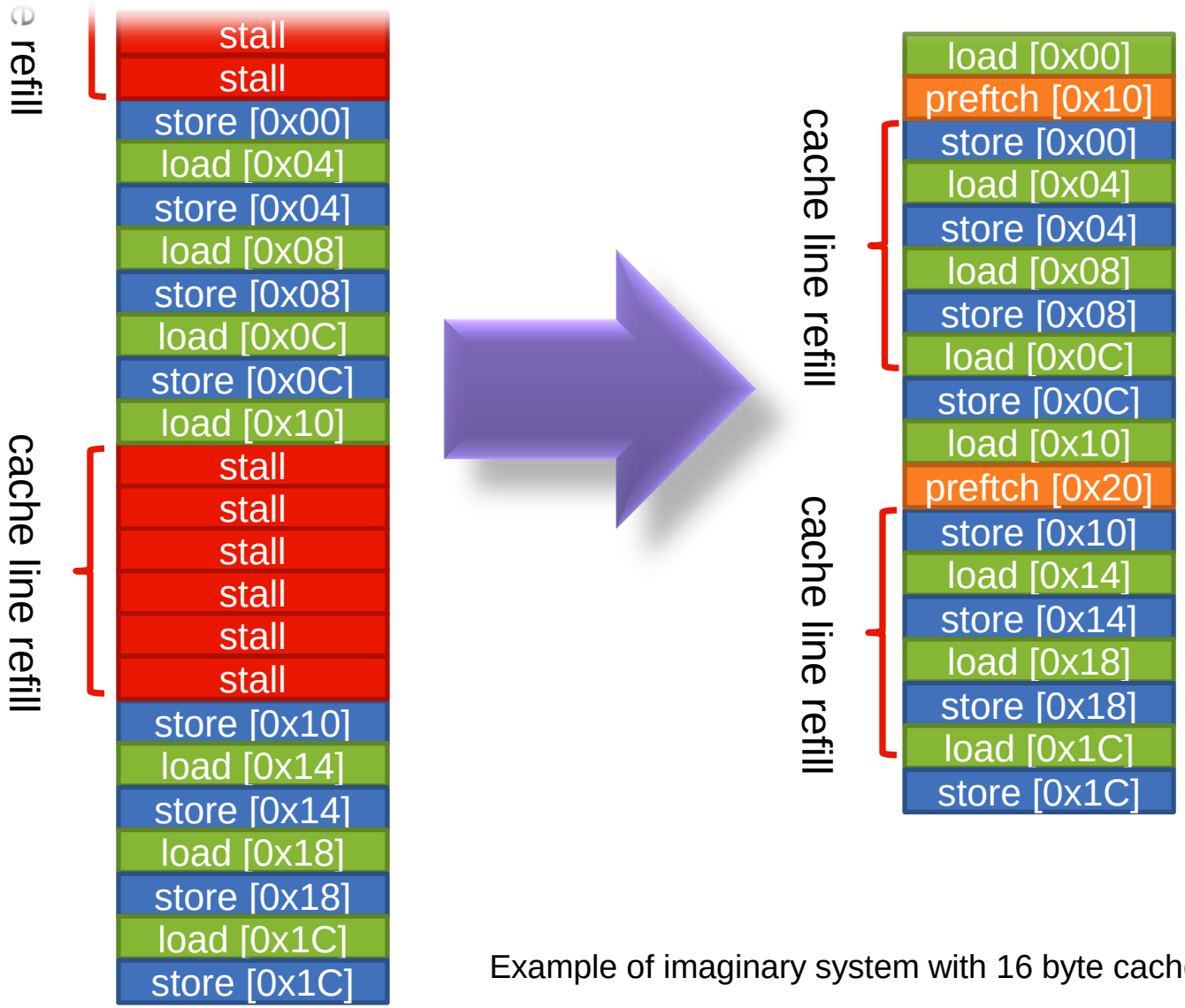


Memory latency causes **stall cycles**



In reality can happen for both load and store, due to **allocate on write** cache line allocation policy

What is prefetching?



Example of imaginary system with 16 byte cache lines

What is prefetching? (2)

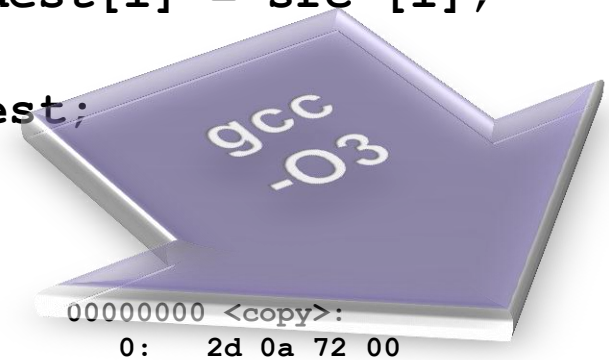
Multiple ways of prefetching

- **HW assisted**
  - CPU tries to recognize patterns, and speculatively fetch more data than requested from memory
- **Compiler assisted**
  - Compiler tries to recognize patterns, and inserts prefetch instructions into the code
- **Manually (using profiling)**
  - SW developer inserts prefetch instructions manually, based on profiling or specific knowledge about an algorithm

Compiler assisted prefetching

Using GCC to generate prefetch instructions

```
long *copy (long *dest, long *src, int size)
{
        int i;
        for (i = 0; i < size; i++) {
                dest[i] = src [i];
        }
        return dest;
}
```




```
00000000 <copy>:
   0:   2d 0a 72 00                    brlt.d      r2,1,2c <copy+0x2c>
   4:   42 21 01 01                    sub         r1,r1,4
   8:   15 26 82 70 ff ff fc ff        add2        r2,-4,r2
  10:   2f 22 82 00                    lsr         r2,r2
  14:   2f 22 82 00                    lsr         r2,r2
  18:   44 71                          add_s       r2,r2,1
  1a:   00 43                          mov_s       r3,r0
  1c:   0a 24 80 70                    mov         lp_count,r2
  20:   a8 20 80 01                    lp          2c <copy+0x2c>
  24:    04 11 02 02                      ld.a        r2,[r1,4]
  28:    04 1b 90 00                      st.ab       r2,[r3,4]
  2c:   e0 7e                          j_s         [blink]
```

Compiler assisted prefetching

Using GCC to generate prefetch instructions


```
00000000 <copy>:
   0:   a9 0a 52 00                 brlt        r2,1,a8 <copy+0xa8>

   4:   a9 0a 52 02                 brlt        r2,9,ac <copy+0xac>

   8:   42 22 45 02                 sub         r5,r2,9
   c:   2f 25 42 01                 lsr         r5,r5
  10:   2f 25 42 01                 lsr         r5,r5
  14:   2f 25 42 01                 lsr         r5,r5
  18:   a4 71                       add_s       r5,r5,1
  1a:   55 21 43 06                 add2        r3,r1,25
  1e:   0a 24 40 71                 mov         lp_count,r5
  22:   00 44                       mov_s       r4,r0
  24:   4a 25 00 00                 mov         r5,0
  28:   a8 20 40 0a                 lp          7a <copy+0x7a>

  2c:   9c 13 06 80                 ld          r6,[r3,-100]
  30:   00 13 3e 00                 prefetch    [r3,0]
  34:   00 1c 80 01                 st          r6,[r4]
  38:   40 24 04 08                 add         r4,r4,32
  3c:   a0 13 06 80                 ld          r6,[r3,-96]
  40:   20 e3                       add_s       r3,r3,32
  42:   e4 1c 80 81                 st          r6,[r4,-28]
  46:   40 25 05 02                 add         r5,r5,8
  4a:   84 13 06 80                 ld          r6,[r3,-124]
  4e:   e8 1c 80 81                 st          r6,[r4,-24]
  52:   88 13 06 80                 ld          r6,[r3,-120]
  56:   ec 1c 80 81                 st          r6,[r4,-20]
  5a:   8c 13 06 80                 ld          r6,[r3,-116]
  5e:   f0 1c 80 81                 st          r6,[r4,-16]
  62:   90 13 06 80                 ld          r6,[r3,-112]
  66:   f4 1c 80 81                 st          r6,[r4,-12]
  6a:   94 13 06 80                 ld          r6,[r3,-108]
  6e:   f8 1c 80 81                 st          r6,[r4,-8]
```

Compiler assisted prefetching

Using GCC to generate prefetch instructions


```
00000000 <copy>:
   0:   a9 0a 52 00
   4:   a9 0a 52 02
   8:   42 22 45 02
   c:   2f 25 42 01
  10:   2f 25 42 01
  14:   2f 25 42 01
  18:   a4 71
  1a:   55 21 43 06
  1e:   0a 24 40 71
  22:   00 44
  24:   4a 25 00 00
  28:   a8 20 40 0a
  2c:   [illegible]
  30:   00 13 3e 00
  34:   [illegible]
  38:   40 24 04 08
  3c:   a0 13 06 80
  40:   20 e3
  6a:   94 13 06 80
  6e:   f8 1c 80 81
```

```
brlt    r2,1,a8 <copy+0xa8>

brlt    r2,9,ac <copy+0xac>

sub     r5,r3,8
lsr     r5,r5
[illegible]
lsr     r5,r5
add_s   r5,r5,1
add2    r3,r1,25
mov     lp_count,r5
mov_s   r4,r0
mov     r5,0
lp      7a <copy+0x7a>

ld      r6,[r3,-100]
prefetch [r3,0]
st      r6,[r4]
add     r4,r4,32
ld      r6,[r3,-96]
add_s   r3,r3,32
st      r6,[r4,-28]
add     r5,r5,8
ld      r6,[r3,-124]
st      r6,[r4,-24]
ld      r6,[r3,-120]
st      r6,[r4,-20]
ld      r6,[r3,-116]
st      r6,[r4,-16]
ld      r6,[r3,-112]
st      r6,[r4,-12]
ld      r6,[r3,-108]
st      r6,[r4,-8]
```

Loop setup
(calculate total number of cache lines to copy)

Prefetch instruction, one per cache line

Unrolled copy loop (one cache line)

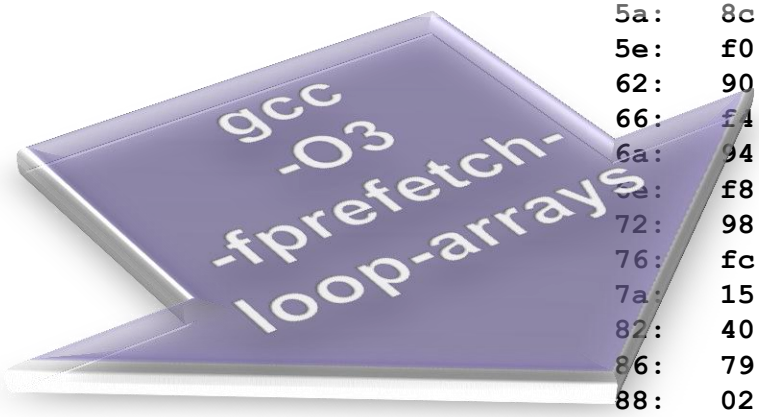
Observations:

- Prefetch stride is 100 bytes ahead*
- Cache line size is 32 bytes
- Prefetch is only emitted for **loads** (in this case)

**) actually prefetch stride is measured in cycles latency by gcc*

Compiler assisted prefetching (contd.)

Using GCC to generate prefetch instructions




```
  46:   40 25 05 02                   add     r5,r5,8
  4a:   e8 1c 80 81                   st      r6,[r4,-24]
  4e:   88 13 06 80                   ld      r6,[r3,-120]
  52:   ec 1c 80 81                   st      r6,[r4,-20]
  56:   8c 13 06 80                   ld      r6,[r3,-116]
  5a:   f0 1c 80 81                   st      r6,[r4,-16]
  5e:   90 13 06 80                   ld      r6,[r3,-112]
  62:   f4 1c 80 81                   st      r6,[r4,-12]
  66:   94 13 06 80                   ld      r6,[r3,-108]
  6a:   f8 1c 80 81                   st      r6,[r4,-8]
  6e:   98 13 06 80                   ld      r6,[r3,-104]
  72:   fc 1c 80 81                   st      r6,[r4,-4]
  76:   15 26 43 71 ff ff fc ff       add2    r3,-4,r5
  7a:   40 25 44 00                   add     r4,r5,1
  7e:   79 61                         add_s   r1,r1,r3
  80:   02 22 7c 01                   sub     lp_count,r2,r5
  84:   29 0a 22 01                   brlt.d  r2,r4,b4 <copy+0xb4>

  88:   00 23 03 00                   add     r3,r3,r0
  8c:   21 0a 80 0f 00 80 00 00       breq    r2,0x80000000,b4
<copy+0xb4>

  94:   a8 20 80 01                   lp      a8 <copy+0xa8>

  98:   04 19 02 00                   ld.a    r2,[r1,4]
  9c:   04 1b 88 00                   st.a    r2,[r3,4]
  a0:   e0 7e                         j_s     [blink]
  a2:   e0 78                         nop_s
  a4:   cf 07 ef ff                   b.d     7a <copy+0x7a>

  a8:   ac 70                         mov_s   r5,0
  aa:   e0 78                         nop_s
  ac:   4a 24 40 70                   mov     lp_count,1
  b0:   f2 f1                         b_s     9c <copy+0x9c>
```

Unrolled copy loop (one cache line)

Do remainder (already prefetched)

GCC options for prefetching

additional options for fine-tuning

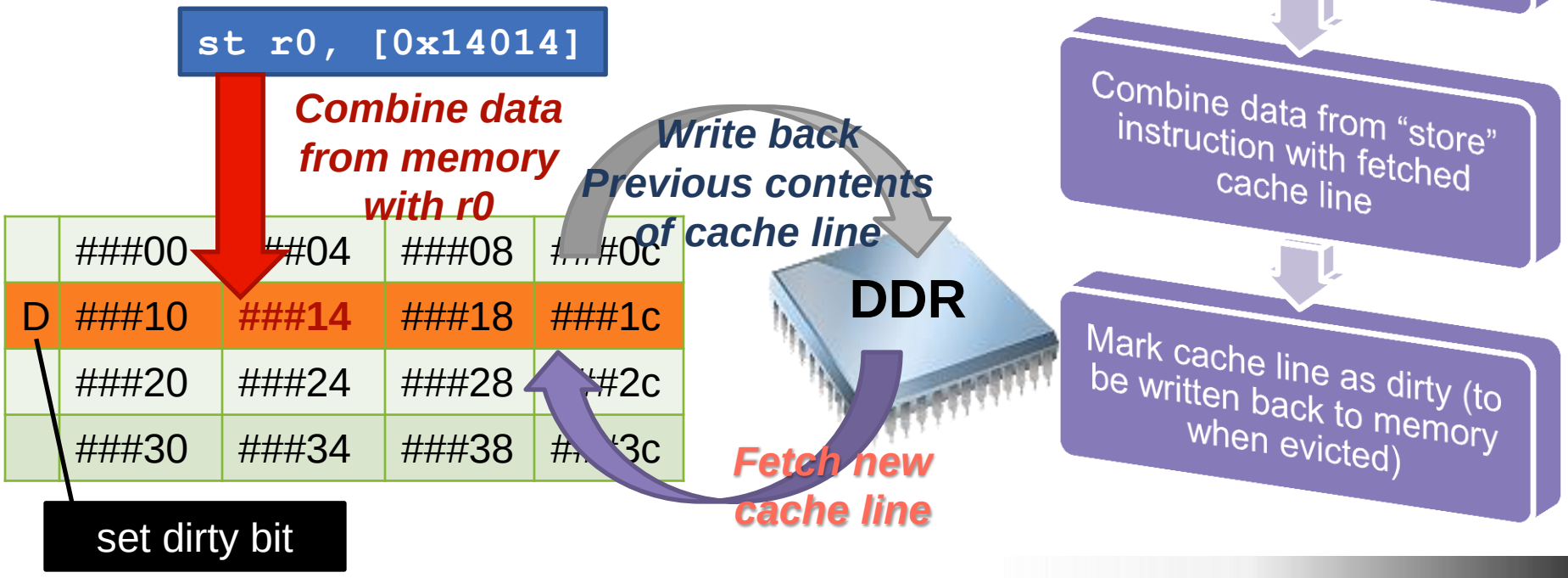
- **-fprefetch-loop-arrays**

parameter	default	unit
prefetch-latency	200	instructions
simultaneous-prefetches	3	
l1-cache-size	64	kiB
l1-cache-line-size	32	bytes
l2-cache-size	512	kiB
min-insn-to-prefetch-ratio	9	instructions
prefetch-min-insn-to-mem-ratio	3	instructions

```
arc-linux-uclibc-gcc -O3 -fprefetch-loop-arrays
                     --param prefetch-latency=128 ./test.c
```

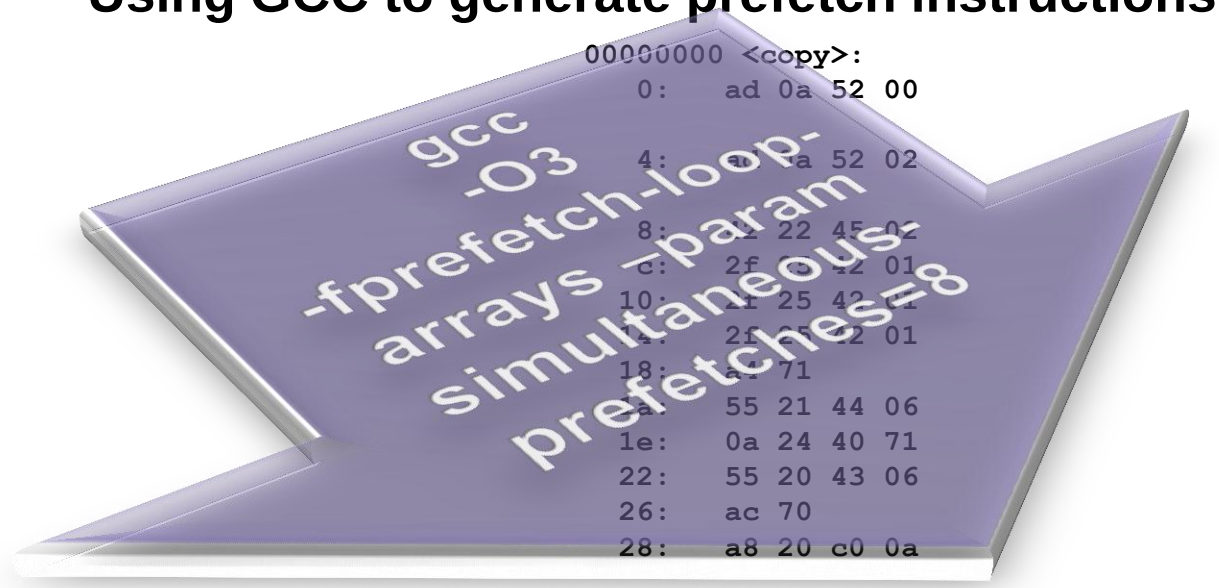
GCC options for prefetching

- Depending on CPU architecture, you may also want to do prefetching for writes



Compiler assisted prefetching

Using GCC to generate prefetch instructions




```
00000000 <copy>:
   0:   ad 0a 52 00
   4:   a0 1a 52 02
   8:   21 22 45 02
   c:   2f 05 42 01
  10:   21 25 42 01
  14:   2f 05 42 01
  18:   a0 71
  1a:   55 21 44 06
  1e:   0a 24 40 71
  22:   55 20 43 06
  26:   ac 70
  28:   a8 20 c0 0a

  2c:   9c 14 06 80          ld        r6,[r4,-100]
  30:   00 14 3e 00          prefetch  [r4,0]
  34:   9c 1b 80 81          st        r6,[r3,-100]
  38:   40 24 04 08          add       r4,r4,32
  3c:   80 14 06 80          ld        r6,[r4,-128]
  40:   00 13 3e 00          prefetch  [r3,0]
  44:   a0 1b 80 81          st        r6,[r3,-96]
  48:   20 e3                add_s     r3,r3,32
  4a:   84 14 06 80          ld        r6,[r4,-124]
  4e:   40 25 05 02          add       r5,r5,8
  52:   84 1b 80 81          st        r6,[r3,-124]
  56:   88 14 06 80          ld        r6,[r4,-120]
  5a:   88 1b 80 81          st        r6,[r3,-120]
  5e:   8c 14 06 80          ld        r6,[r4,-116]
  62:   8c 1b 80 81          st        r6,[r3,-116]
```

```
brlt    r2,1,ac <copy+0xac>

brlt    r2,9,b0 <copy+0xb0>

sub     r5,r2,9
lsr     r5,r5
lsr     r5,r5
lsr     r5,r5
add_s   r5,r5,1
add2    r4,r1,25
mov     lp_count,r5
add2    r3,r0,25
mov_s   r5,0
lp      7e <copy+0x7e>
```

By increasing the *simultaneous-prefetches* parameter, gcc also emits prefetch instructions for writes.

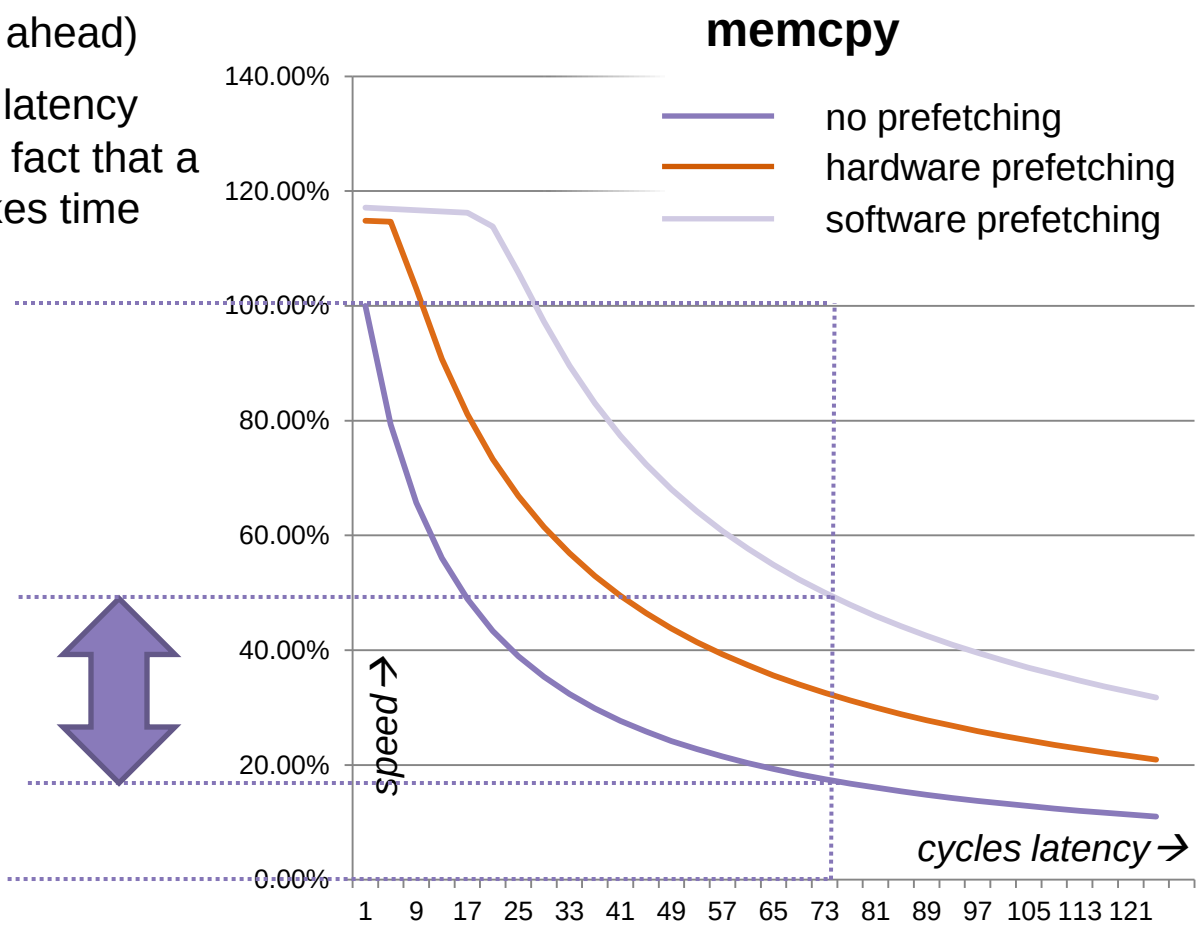
Other prefetch methods

- Manually, using prefetch builtin:
  - `__builtin_prefetch(ptr)`
  - `__builtin_prefetch(ptr,1)` (prefetch for writing)
- Pointer is allowed to be NULL; no exception / segfault is supposed to happen.
- Prefetching is used inside Linux kernel:
  - Inside the memory allocator (slab/slub), inside RCU trees
- **WARNING:** prefetching and DMA can cause trouble, as dma_map_single() calls cause code to assume that certain data is not in the cache. **Make sure that you don't prefetch beyond DMA buffer boundaries!** *(depends on I/O coherency)*
- Hardware prefetching should be transparent;
  - HW recognizes consecutive reads/writes and may speculatively fetch adjacent lines
  - Takes a couple of loop iterations before HW can recognize a pattern

Memcpy performance with prefetching

- Simulation with:
  - HW prefetcher (one cache line ahead)
  - SW prefetching (512 bytes ahead)
- **NOTE:** Even without memory latency prefetching is useful due to the fact that a cache line refill in itself also takes time

More than twice as fast at 75 cycles latency with prefetch instructions



Improving branch prediction

- Linux defines likely() and unlikely() macros:
  - `#define likely(x)   __builtin_expect(!!(x), 1)`
  - `#define unlikely(x) __builtin_expect(!!(x), 0)`
- The gcc built-ins affect:
  - Scheduling of code (i.e. likely means branch not taken);
  - Depending on architecture they may give hints to branch predictor
- **WARNING:** While (x) may be true, (x) isn't necessarily 1 (i.e. 2 is also true). Therefore the Linux macro's use !!(x).
- **WARNING:** Don't make things worse; if you add a hint, make sure it's correct! (use actual profiling data)

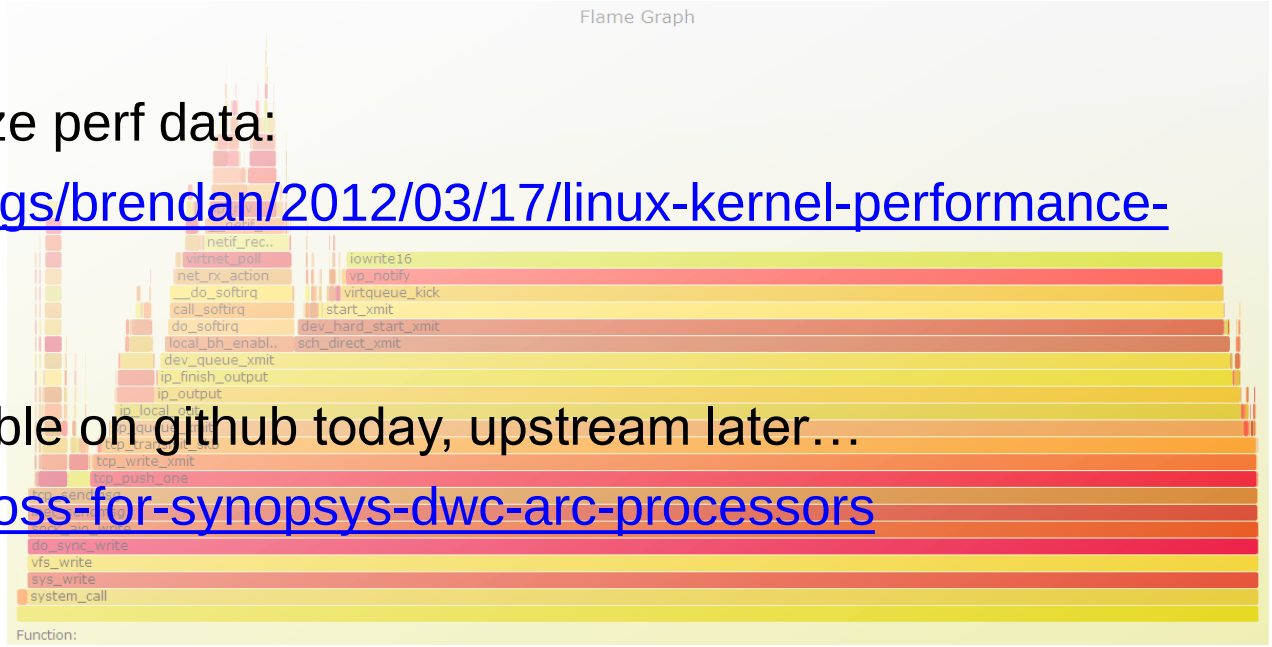
Further reading

- Paper about optimizing a rasterization library; also talks about using prefetching:
- http://ctuning.org/dissemination/grow10-03.pdf

- Blog entry about likely/unlikely
- http://blog.man7.org/2012/10/how-much-do-builtinexpect-likely-and.html

- Cool way to visualize perf data:
- http://dtrace.org/blogs/brendan/2012/03/17/linux-kernel-performance-flame-graphs/

- Perf for ARC available on github today, upstream later…
- https://github.com/foss-for-synopsys-dwc-arc-processors



Thank you!

Questions?